2017

Design and Demonstration of a Physical, Multi-Agent Autonomous Controller Testbed

Eric A. Nees
Wright State University

Design and Demonstration of a Physical, Multi-Agent Autonomous Controller Testbed

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering

by

Eric A. Nees
B.S.E.E., Rose-Hulman Institute of Technolgy, 2008

2017
Wright State University

Wright State University
GRADUATE SCHOOL

August 23, 2017

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Eric A. Nees ENTITLED Design and Demonstration of a Physical, Multi-Agent Autonomous Controller Testbed BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Electrical Engineering.

Zachariah E. Fuchs, Ph.D.
Thesis Director

Brian Rigling, Ph.D.
Chair, Department of Electrical Engineering

Committee on
Final Examination

Zachariah E. Fuchs, Ph.D.

Josh Ash, Ph.D.

John C. Gallagher, Ph.D.

Robert E.W. Fyffe, Ph.D.
Vice President for Research and
Dean of the Graduate School

ABSTRACT

Nees, Eric A. M.S.E.E., Department of Electrical Engineering, Wright State University, 2017. *Design and Demonstration of a Physical, Multi-Agent Autonomous Controller Testbed.*

Navigation and control algorithms are often tested in a simulated environment before being deployed in physical systems. Although simulated environments provide a controlled setting to carefully evaluate performance, the designed scenarios are sometimes unrealistically ideal and may unintentionally omit circumstances or unmodeled interactions. This thesis presents the design, implementation, and practical demonstration of a physical testbed that enables the testing of multi-agent autonomous strategies in hardware on a small scale. Testing the algorithms at scale allows real-time exploration of the interaction and performance of both human and autonomous algorithms under non-ideal conditions while avoiding the costs and risks of full-scale, deployed systems. Presented in the following is a detailed robot design for this explicit purpose, and the overall design of the testbed system including software. A dynamic game is evaluated using the described system, and the results are presented.

Contents

List of Figures

List of Tables

Acknowledgment

This project wouldn't have been possible without the support of my advisor, Dr. Fuchs, whose expertise and guidance was invaluable. The initial concept was his, and for that I owe him this whole project. I need to thanks my colleague, Pavlos Androulakakis, who graciously offered me the use of his autonomous controllers for the demonstration section of this project and spent his time supporting their use. I would also like to thank my committee members for sacrificing a few nice summer days to read my paper and participate in my defense. Finally my wife, Lacey, for her patience and support on days when I was either too wrapped up in my own thoughts or, worse, engaging her with some kind of unsolicited technical monologue.

Bridging the Gap

1.1 The Problem

Humans, in the performance of even the most mundane daily tasks, demonstrate a wide variety of problem solving and cost-function-estimation abilities. We take these abilities for granted, and we imagine that they might be simple to reproduce because we observe that many animals can perform these kinds of operations quite effectively[8]. Unfortunately, simply observing that a behavior happens and understanding it on an intuitive level does not constitute an algorithmic solution. The math required to solve something like a simple navigation or path-finding problem is rarely trivial. The inverse is also true; as they increase in complexity, algorithms and control strategies (all systems really) tend to become less intuitive in terms of what behavior they will produce. Sometimes the best way to learn about a system is to interact with it.

1.2 A Physical Testbed

The following sections of this document describe the design and implementation of a physical testbed for exploring the behavior of and interaction between humans and autonomous algorithms. The testbed isn't designed to perfectly reproduce any particular real-world or simulated environment. The testbed is somewhere in between: more controlled and ideal

than most real-world environments, but entirely tangible in a way that makes real-time physical interaction possible. The hope is that this testbed will enable innovative research and experimentation, in a way that's accessible to even the casual observer.

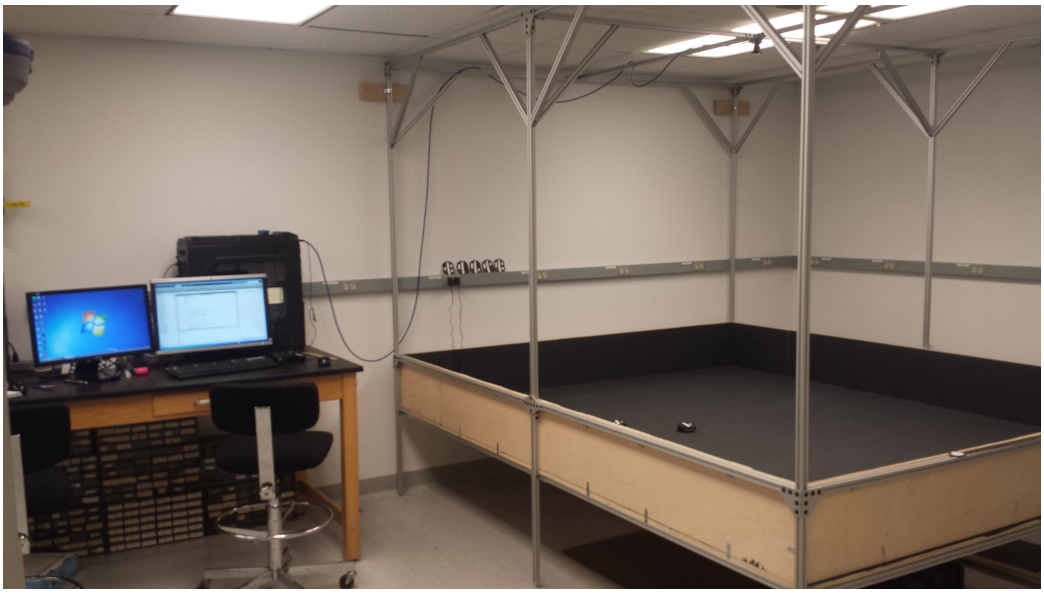


Figure 1.1: Testbed with control computer

Design of the Testbed System

2.1 Overall Architecture

The proposed testbed requirements describe a flat and level area, indoors, with a camera system to track robot positions and a central computer to perform all the control and logging functions. The system should be able to support multiple robots, autonomous and otherwise, in any combination. See Figure 2.1.

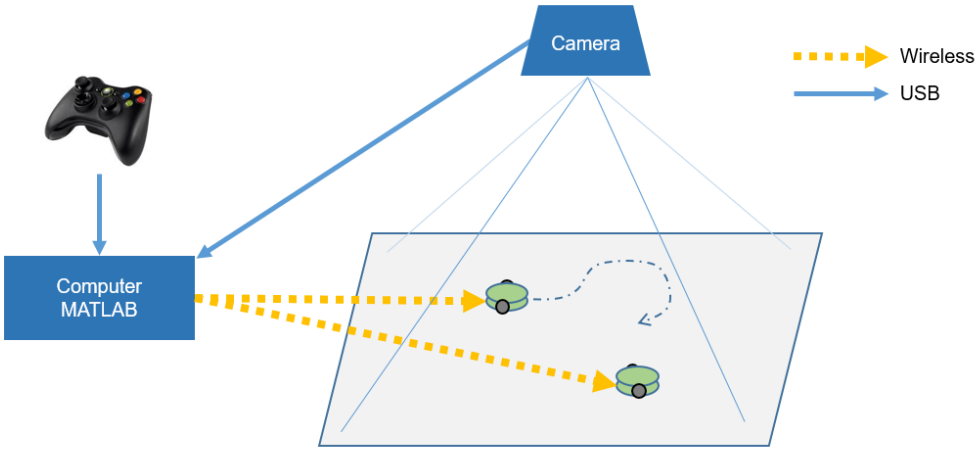


Figure 2.1: Testbed Setup

The sensor suite required to track robot position etc would be too large and complex to implement on each robot individually, so it's an easy decision to use a separate computer and camera vision system to track the robots globally. The same computer runs the control

algorithms for each robot and generates commands. Offloading the control algorithms from the robots' onboard controller to the central computer has several advantages:

- Algorithms can be developed and run in MATLAB instead of C or C++
- Much higher processing capability
- No need to flash new firmware into each robot every time the algorithm changes

Since robot control commands are generated on the computer, a wireless communication link is required to send these commands to individual robots.

In this way, the general testbed architecture is decided: a central computer for performing algorithms, a camera to detect robot positions, a computer to perform the control and logging functions, and wireless links for robot control.

2.2 Computer and Software

For the control computer, MATLAB was chosen as the language for the main program. Advantages of this setup include the ability to leverage existing toolkits for computer vision functions, and the ability to drop existing controller code directly into the program. Since one of the goals of this project is to create a testbed that will be usable in the future, easy interface to existing and developmental algorithms is essential.

After initializing the camera, wireless interface, and log files, the software starts a loop which runs continuously. This loop performs the following steps, in order, before repeating:

- Acquire an image
- Correct the image for lens distortion
- Estimate Robot State (position, orientation)

- Calculate and issue a command for each robot
- Record all robot parameters and commands to a log file
- Plot the image and some command details to the screen

This process is repeated at approximately 20hz. In MATLAB, this process is contained within a timed event which the software attempts to call at 20Hz. If the process execution time is longer than 50ms for any reason, it might not be possible to achieve 20Hz consistently. Even if the process typically takes less than 50ms, computers with high level operating systems like Windows sometimes decide to share CPU time in a way that can cause unexpected variations in processing speed for a given application. In order to achieve smooth and consistent control, it's necessary to mitigate the effects of these variations in time-step size. The software accomplishes this by measuring the real execution time of each iteration, so the effects of small irregularities in time-step size are mitigated. At the time of writing, the execution speed of the MATLAB image processing functions was the bottleneck in terms of loop speed. Higher rates would likely be achievable using a more specialized setup, which is beyond the scope of this paper.

2.2.1 Acquire an Image

Image acquisition is performed by MATLAB using the MathWorks Image Acquisition Toolbox. The camera used on the test bed is a Chameleon 3 from FLIR Integrated Imaging Solutions Inc., formerly Point Grey Research, equipped with a 3.5mm C Series Fixed Focal Length Lens with a 94 degree field of view and a USB3 interface for power and data. Fortunately the manufacture provides drivers which allow this camera to interface easily with MATLAB.

While there are a plethora of configurable options for the camera and driver, only a small set of them is required to achieve good results. Figure 2.2 shows suitable camera configuration code for this testbed. The most important parameter is actually the frame

rate; it must not exceed the rate that's expected by the software. If the camera frame rate exceeds the software loop rate, the software will sometimes miss a frame entirely. This causes a sudden jump of 2x the normal δt between images, but with a 1x loop δt. The result is erroneous velocity results and what appear to be discontinuities in what are otherwise smooth signals.

```
1 vidobj = imaq.VideoDevice('pointgrey', 1, '
     F7_Mono8_2048x1536_Mode0');
2 vidobj.ROI = [323 172 1529 1201];
3 vidobj.DeviceProperties.Exposure = -0.4;
4 vidobj.DeviceProperties.FrameRate = 20;
5 frame = step(vidobj);
```

Figure 2.2: Camera Initialization

Exposure and Region of Interest can be adjusted to match the needs of the test. The camera is capable of color images as well, but color imaging was ultimately determined to be unnecessary for the work described in this paper.

2.2.2 Perform Lens Distortion Correction

The wide field of view provided by the selected lens allows the camera to capture a large surface area without being excessively far away. The downside to this is that the image is significantly distorted, especially near the edges. This distortion makes linearly scaling pixel positions directly to real-world units impossible. An additional calculation to remove the lens distortion is required.

Fortunately, MATLAB's Computer Vision Systems Toolbox provides a function to do this. By imaging a checkerboard pattern of known dimensions from multiple angles, the toolbox can calculate a set of lens distortion parameters. Figure 2.3 shows the MathWorks camera calibrator app in use with the objective camera and lens. These parameters can then be used to process the image and remove lens distortion.

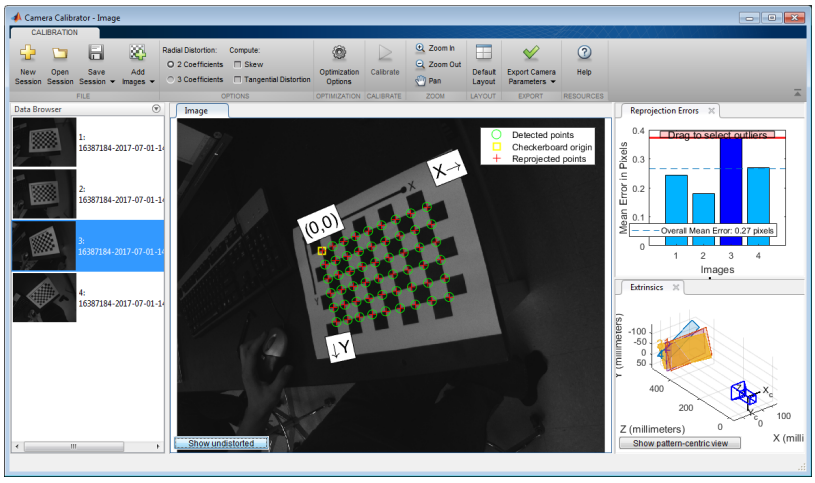


Figure 2.3: One of several Calibration Images used to calculate Lens Distortion Parameters

The primary type of distortion evident in the image is positive radial, or barrel distortion. Figure 2.4 demonstrates how barrel distortion causes a uniform grid to project differently. Compare this to the left half of figure 2.5, which shows an example of a distorted image from the actual testbed, and notice the characteristic outward-bending lines. The effects of radial distortion are more severe towards the edges of the field of view.

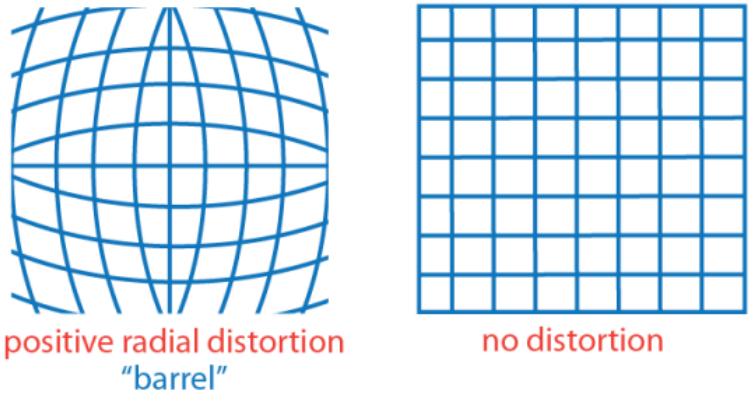


Figure 2.4: Planar representation of positive radial distortion[6]

Equation set 2.1[6] can be used to re-project the distorted image back onto a uniformly scaled orthogonal grid. The coefficients k_1 and k_2 are generated by the calibrator app shown in Figure 2.3. MATLAB's Computer Vision Systems Toolbox function *undistortImage* uses these coefficients and performs the transformation described in equation set 2.1 to correct

the image.

$$
\begin{aligned}
x_{distorted} &= x(1 + k_1 r^2 + k_2 r^4) \\
y_{distorted} &= y(1 + k_1 r^2 + k_2 r^4) \\
&\text{where} \\
r^2 &= x^2 + y^2
\end{aligned}
\tag{2.1}
$$

Figure 2.5 shows an example of lens correction in action on the testbed. The two straight metal bars in the left image are distorted and appear bent. The distortion is particularly apparent at the outer edges of the image. The image on the right has been corrected using MATLAB's Computer Vision Systems Toolbox *undistortImage* and the camera parameters generated by the calibrator app. It's clear that the second image exhibits significantly less distortion.

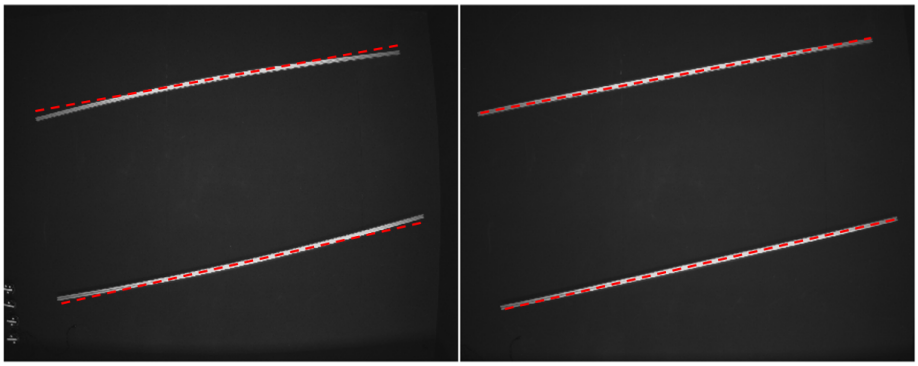


Figure 2.5: Two straight metal bars, before and after lens correction

Once the distortion is removed, the image pixel grid can be linearly related to the real-world grid. That is, we can calculate a scale factor from pixels to millimeters. This holds true because everything we're imaging from here on out is effectively on a 2-dimensional plane which is at a fixed distance from the camera.

2.2.3 Estimate Robot States

Each robot has a black top plate which, from the perspective of the camera, blends into the black surface of the table. The top plate has a series of white marks on it which are used to identify the robot's position, orientation, and identity. A digram of the top plate is shown in Figure 2.7.

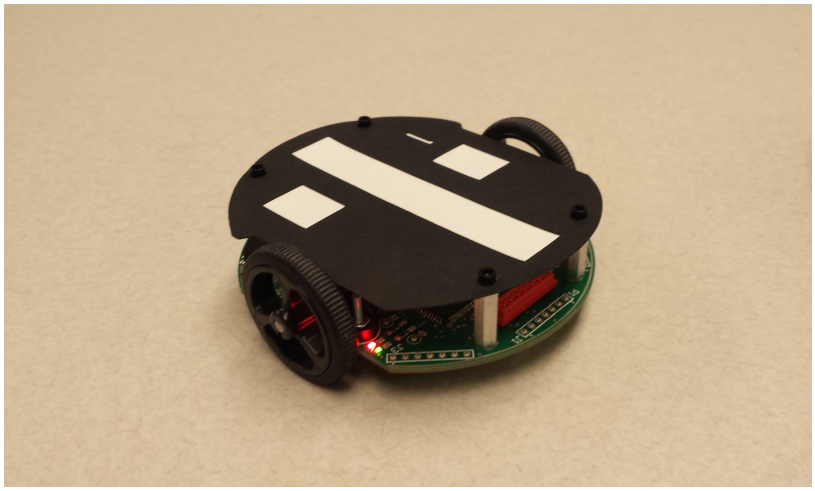


Figure 2.6: Robot, as built, with to plate showing typical identification markings

We begin the state estimation process by thresholding the corrected image into black and white, and running the result through the *regionprops* function from the MathWorks Image Processing Toolbox. This function identifies contiguous regions of white pixels and calculates some parameters which describe each region. For each region, the *regionprops* function returns parameters for the region's size, aspect ratio, minimum bounding ellipse, and location within the image.

Robot position is estimated using the Position Indicator, which is the largest indicator marking on each robot. The Position Indicator is a long rectangle located in the center of the top plate (see figure 2.7). The software searches for this indicator among all the regions identified by the *regionprops* function based on size and aspect ratio. Filtering by size and aspect ratio allows the software to reject the other smaller markings on the robot, as well

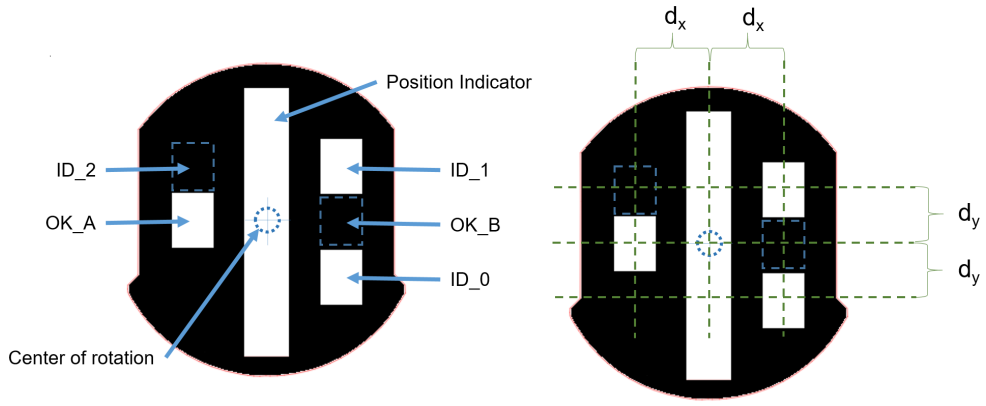


Figure 2.7: Robot Top Plate, with marking locations identified (robot ID = 3)

as other objects that appear in the image, including dust and other small particles which appear bright against the black background. The *regionprops* function returns an x-y pair specifying the center of mass of the Position Indicator, and a orientation result which is part of the description of the minimum bounding ellipse. The x-y pair can be used directly as the robot position, but the orientation value requires more processing.

There are also several other markings on the robot. The two Orientation Key markings (OK_A and OK_B, figure 2.7) are used to distinguish the front of the robot from the back, while the Identification Markings (ID_0 through 2, figure 2.7) are used to uniquely identify the robot using a binary value 0-7. The relative locations of these markings with respect to the Position Indicator are constant, meaning that once we know where the Position Indicator is and what orientation it has, we can locate the other markings as well. The relative positions of the markings are defined in Figure 2.7.

The Position Indicator's orientation value describes the angle of the major axis of the minimum bounding ellipse of the region (shown in red in figure 2.9) as measured against the x axis, which means that while it correctly identifies the axis of the robot, it fails to distinguish the front of the robot from the back. The Orientation Key (OK) marking fields are used to discern the left of the robot from the right, and thus the front from the back. Marking OK_A is always White and always on the left side of the robot, while OK_B is always Black and always on the right side (aft looking forward). Knowing the center of the

Position Indicator, and the major axis of the shape, we can calculate the expected positions of OK_A and OK_B:

$$\begin{aligned} x_{OK\_A} &= x_c + d_x sin(\theta_P) \\ y_{OK\_A} &= y_c + d_x cos(\theta_P) \end{aligned} \tag{2.2}$$

$$\begin{aligned} x_{OK\_B} &= x_c - d_x sin(\theta_P) \\ y_{OK\_B} &= y_c - d_x cos(\theta_P) \end{aligned} \tag{2.3}$$

where d_x is the distance from the center of the Position Indicator to the center of the OK markings in each direction, and x_c, y_c, and θ_P are the positions and orientation of the Position Indicator as reported by the *regionprops* function.

Once we've determined where the markers are, it's a simple matter of checking which one is black vs white. These locations are each marked with a green 'x' in figure 2.9. If OK_A is detected as white and OK_B as black, as in the left half of Figure 2.8, we can use the orientation value directly. Otherwise we know the robot is facing the opposite direction, and we need to add π to the orientation value. In figure 2.9, the robot's forward direction has been marked with a yellow circle.

In a similar way, the expected positions of the three ID markers can be calculated as well:

$$\begin{aligned} x_{ID\_0} &= x_c - d_y sin(\theta_{PC}) - d_x cos(\theta_{PC}) \\ y_{ID\_0} &= y_c + d_y cos(\theta_{PC}) - d_x sin(\theta_{PC}) \end{aligned} \tag{2.4}$$

$$\begin{aligned} x_{ID\_1} &= x_c - d_y sin(\theta_{PC}) + d_x cos(\theta_{PC}) \\ y_{ID\_1} &= y_c + d_y cos(\theta_{PC}) + d_x sin(\theta_{PC}) \end{aligned} \tag{2.5}$$

$$\begin{aligned} x_{ID\_2} &= x_c + d_y sin(\theta_{PC}) + d_x cos(\theta_{PC}) \\ y_{ID\_2} &= y_c - d_y cos(\theta_{PC}) + d_x sin(\theta_{PC}) \end{aligned} \tag{2.6}$$

where θ_{PC} is the orientation of the Position Indicator as corrected by the above analysts of

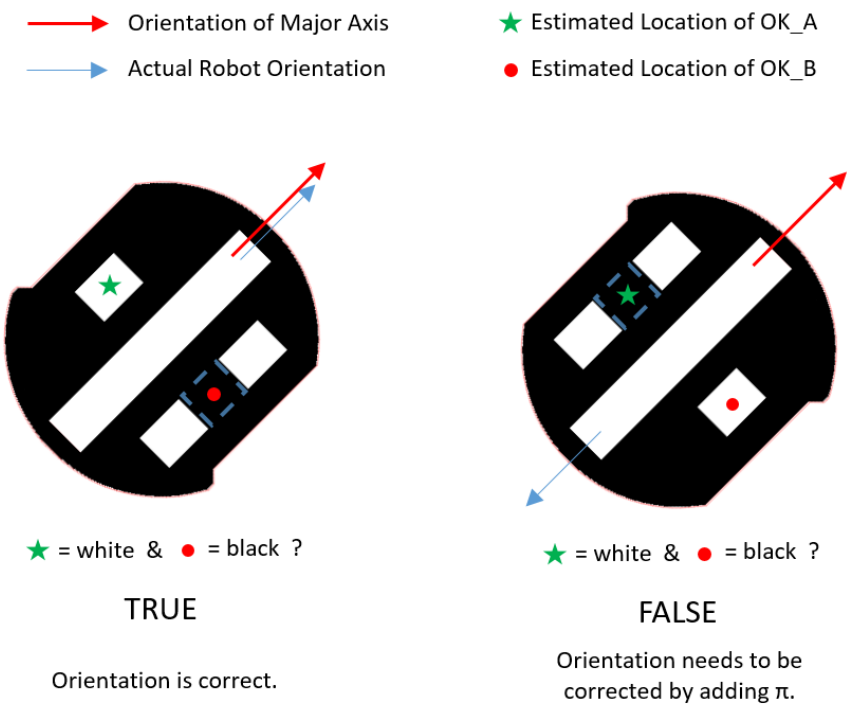


Figure 2.8: Example use of Orientation Key markings

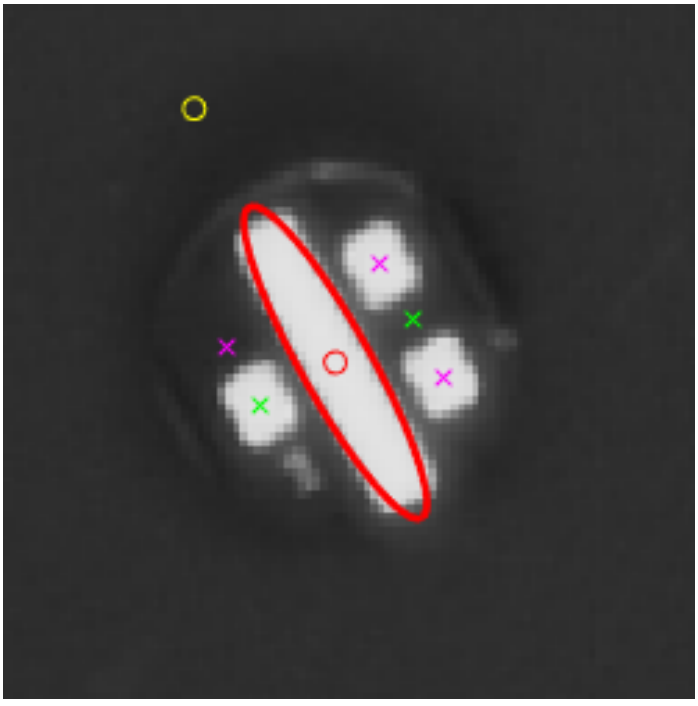


Figure 2.9: Robot as imaged, with key features identified and plotted by the software

the OK markings.

Positions ID_0, ID_1, and ID_2 combine to form 3 bits of a binary number which represent the robot's identification. White represents a 1, and black represents a 0. These three markings match a set of dip switches on the underside of the robot which are read by the local microcontroller. See section 3.6 for a description of how this hardware works. In this way, the robot can identify itself and the computer can identify it as well.

2.2.4 Issue Commands

Generating commands via some control strategy algorithm is easily accomplished within the computer, but for human interaction a user interface becomes necessary. A standard xbox controller was chosen for it's compatibility with MATLAB, and the fact that it and similar interface controllers are common and well-understood by many people.

The mapping of the controller inputs to the robot commands is entirely flexible, but for the purposes of the tests described in the later sections of this paper, the left thumb stick was mapped to the robot's wheel speed commands in a way that allowed forward and reverse motion, turns of any radius down to zero, and full stop. The y axis of the stick (forward and backwards) was mapped to an average wheel speed, and the x axis was mapped to a delta wheel speed. By summing the average and deltas on each wheel, separate left and right speed commands are generated. The following example code performs this operation:

```
1 [axis, buttons, povs] = read(joy);
2 if (abs(axis(2))<0.20) axis(2) = 0; end %deadband
3 if (abs(axis(1))<0.20) axis(1) = 0; end %deadband
4 left_cmd = -20*axis(2) + 8*axis(1);
5 right_cmd = -20*axis(2) - 8*axis(1);
```

Figure 2.10: Joystick command mapping

Line 1 of figure 2.10 reads the position of the joystick as a 2-element array into the *axis* variable. Each value represents one of the two axes of motion, as a number from -1 to

1. Lines 2 and 3 implement a deadband around zero, to ensure that the robot stops moving when the user centers the thumbstick. The constants on lines 4 and 5 effectively scale the commands such that the robot moves in the expected direction at high top speed and turn rate. These constants can be adjusted in magnitude in order to change the performance of the robot.

Alternative, perhaps more complex mappings might be more suitable for other kinds of tests, including mappings which artificially limit speed or turn rate, or disallows backwards motion for example.

Once a robot command has been generated, it is transmitted to the robot via a transparent, wireless RS-232 link (described in greator detail in section 3.6). Commands are structured and sent as described in figure 2.11. Each robot's firmware responds only to commands for that particular ID, making it possible to broadcast control packets to many robots and have each one respond appropriately.

```
1  %format for movement command is as follows: #ABBCC*
2  % where # is the start charecter
3  % where A is the target robot ID, a number 0 to 7
4  % where BB is the command* for the right wheel speed
5  % where CC is the command* for the left wheel speed
6  % where * is the stop character.  This character triggers
      the
7  % robot to act on the command.
8  % *wheel speed command is offset by 30 counts.
9  cmd = ([sprintf('#'),...
10 sprintf('%01i',round(ID(ind))),...
11 sprintf('%02i',round(right_cmd(ind)+30)),...
12 sprintf('%02i',round(left_cmd(ind)+30)),...
13 sprintf('*')]);
14
15 %send command
16 fprintf(app.XBee,cmd);
```

Figure 2.11: Format and transmitting of robot movement command

2.2.5 Log States and Commands

All robot states and commands are logged at every timestep in order to facilitate playback, post processing, and analysis. The log file format is a simple Comma Separated Value table, which makes it easy to import into various programs for analysis. In particular, MATLAB function *readtable* makes this process easy. The first column of the file is a timestamp for each line, making it possible to evaluate time-dependent behaviors. The following columns contain robot states (x, y, and angular position) and commands as sent (left and right wheel speed).

The software automatically keeps a running tally of how many times it's been run and stores the result in a binary file. This tally system creates a series of sequential integers which can be used to uniquely identify particular log files. A new log file is created for every run, and saved with a filename which contains a timestamp and the unique identifier. This feature enables better documentation of experiments by allowing users to refer to log files by concise and unique integer identifiers.

2.2.6 Display Visualizations

The final step of the software loop is to plot some key data as an overlay to the camera image inside a MATLAB application window. By plotting measured robot positions and projected paths of travel in real time, the operator is able to quickly evaluate the behavior of the whole system. A typical example of the contents of the application window is shown in figure 2.12.

Additional lines, annotations, and plots are created easily and overlaid as required. In the future, a more elaborate Graphic User Interface may enable additional control or visualization features.

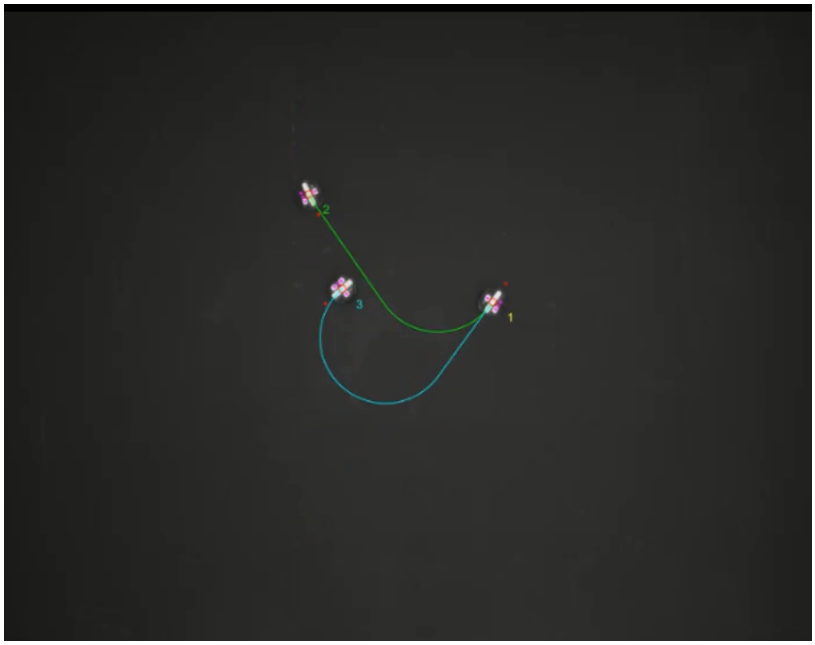


Figure 2.12: Typical application window content

Design of the Robot

3.1 Robot Architecture

In the interest of conserving valuable indoor real estate, the overall size of the area available was limited to about 12ft by 8ft. Many previously described robotic testbeds are quite large by comparison, perhaps mainly out of the need to accommodate available robots of a certain size. This space limitation presented a problem in terms of actually having enough space to position and maneuver multiple robots. Implementation therefore required a relatively small robot.

An ideal robot for this testbed might be a infinitesimally small dot, similar to the zero-dimensional size of a simulated robot. In reality, the lower bound for the size of the required robot is a function of some very practical considerations including:

- Construction and maintenance - The robot will be built by hand, so excessively small parts may be too difficult to handle
- Physical durability - The robot will be handled repeatedly, and needs to survive crashing into walls, etc
- Cost and availability - Commodity components only come so small before their complexity or cost puts them outside our scope or budget

A brief analysis of off-the-shelf options is presented in table 3.1, with the in-house-designed robot described in this report referred to as “G8”. Of the robots which directly

satisfy the general requirements, the G8 design is the least expensive by a wide margin. In truth, the G8 design is based on the 3pi platform from Pololu[3] and takes inspiration from that robot's mechanical systems in particular.

	3pi	**Elisa-3**	**e-Puck**	**G8**	**Khepera IV**
Size	9.5cm	5cm	7cm	9cm	14cm
Price	$100	$390	$900	$150	$3000
Wheel Speed Feedback	no	no	yes	yes	yes
Integrated Wireless	no	yes	yes	yes	yes
Expandable	yes	yes	yes	yes	yes
Meets Requirements?	no	no	yes	yes	yes

Table 3.1: Robot Alternatives[9]

The final 9cm size of the G8 robot was of course a function of the trades described in the following sections. In the end, this was largely driven by the size of available, high quality drive motors which are described in section 3.5. Concerning the time and effort required to construct the robot, it was determined that the G8 robot could be built by a relatively inexperienced graduate student in about 3 hours. This level of effort was considered acceptable for the use-case presented in this document. If in the future a large number of robots were required, the best course of action would likely be to send the design to a company that specializes in small runs of custom products[10].

Figure 3.1 shows a high level overview of the selected G8 robot architecture, with major systems and connections identified.

3.2 Physical Geometry and Printed Circuit Board

It's desirable for the robot to have a tight or zero turn radius. Differential drive was selected to achieve this, and for the sake of simplicity. Ideally, the robot would be able to rotate around it's own center without any part "swinging out" and interfering with nearby objects.

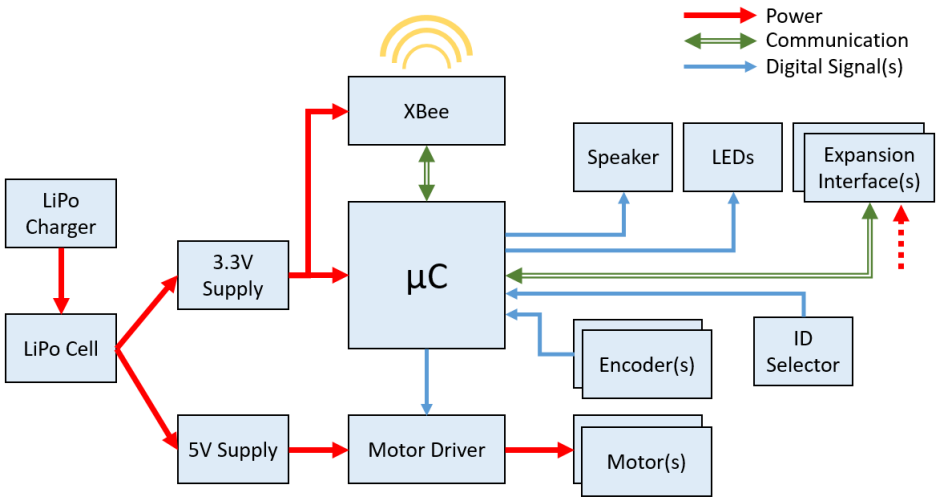


Figure 3.1: Robot Architecture

Accordingly, a round chassis shape was chosen, with the drive wheels placed on the outer edges near the center of the robot's axial length. In order for the robot to be stable on two wheels, a third point of contact with the surface was required. A spherical caster was placed at the rear of the chassis. The selected caster is solid steel in order to shift the center of mass backwards and help ensure that the center of gravity is between the wheels and the caster (Figure 3.2).

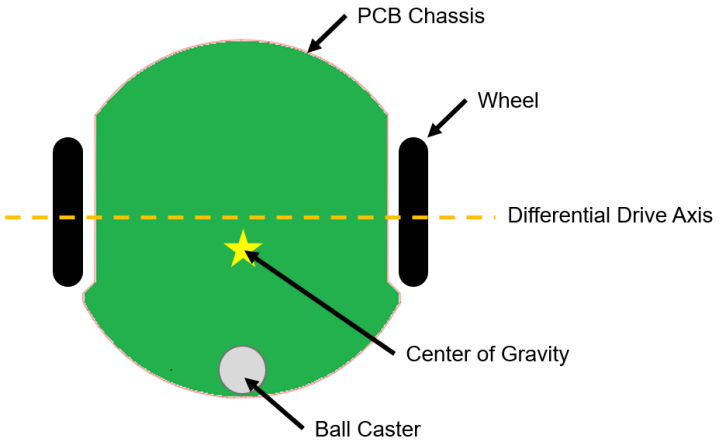


Figure 3.2: Top down view showing wheels and caster in relation to the center of gravity

In order to reduce complexity and size, it was decided to make the chassis of the

robot out of a 1.6mm thick PCB. A fiberglass composite PCB is well suited to function as a chassis for a small robot due to its high strength and rigidity, and the fact that it is manufactured to tight dimensional tolerances. This chassis PBC is dual-purpose in that it incorporates all the pads and electrical traces required for the robot circuitry, making this configuration extremely efficient in terms of size and complexity. Significant cost savings are realized by implementing this PCB using only 2 layers.

The schematic, PCB, and spacial models were developed using CircuitMaker by Altium. CircuitMaker is a powerful integrated design environment for creating PCB designs. It combines schematic capture, layout, and spacial design models like the one shown in figure 3.3. The detailed final design is posted online, and available for use by interested parties[7].

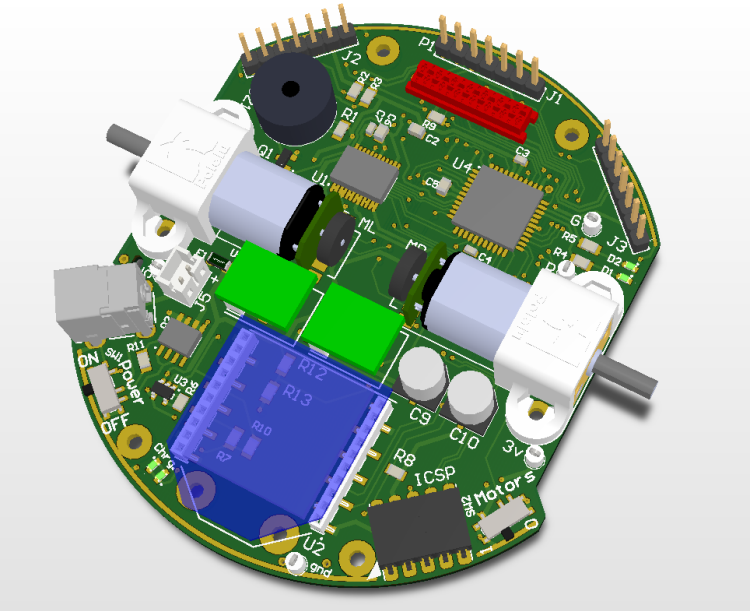


Figure 3.3: Robot PCB Model[7]

At this point, it begins to be apparent that the small size of the vehicle has some significant advantages. Where a larger vehicle might require a more complex chassis, this small robot has no dedicated chassis to speak of and the differential drive works with a simple unlubricated spherical caster. The weight on the caster is minuscule (less than 20g),

which greatly mitigates any concerns about friction or wear for either the robot or the surface on which it's driving.

3.3 Microcontroller

The robot required an onboard microcontroller capable of performing all the required tasks, while also maintaining a suitably small size, power consumption, and ease of handling. In the interest of miniaturization, it was decided to directly integrate a microcontroller IC rather than a pre-built module.

Eliminated from consideration were any microcontrollers which require external memory, clocks, etc. Additionally, any with packages unsuitable for hand-assembly on a 2 layer circuit board were eliminated. A 32 bit microcontroller from Microchip was selected (see Appendix A). This chip has a feature which allows remapping peripheral functions between pins, greatly reducing trace routing problems on 2 layer PCBs. This chip can be programmed while in-circuit by interfacing a tool such as the PICKit3 programmer to a header that's been designed onto the robot.

3.4 Firmware

The robot firmware was written in C using the MPLABX IDE from Microchip Corporation. The firmware goes through an initialization phase before dropping into an infinite loop trap and allowing the operational functions to be entirely interrupt-driven. The firmware uses two interrupts sources to trigger actions.

The high priority interrupt runs based on a timer which is configured for 50Hz. Its primary function is to regulate wheel speed. Each time this interrupt executes it checks the value of the counter peripherals which monitor the encoders, and uses those values to calculate instantaneous wheel speeds. The actual instantaneous speeds and setpoints are

fed into a integral controller with a strong feed-forward compensation in order to generate duty-cycle values for the PWM modules which feed the motor controller (further described in section 3.5).

The lower priority interrupt is triggered when a byte arrives from the XBee at the serial interface. Each time a byte arrives, the firmware checks its value and appends it to the end of a multi-element buffer. Start characters as identified in table 3.6 trigger a reset and clear of the internal buffer. Stop characters as identified in table 3.6 trigger the firmware to process the contents of the buffer. Any other character is simply added to the buffer and assumed to be part of the contents of the message. Since this process is interrupt-driven it runs as often as required in order to keep the internal receive buffer from overflowing.

3.5 Drivetrain

For the differential drive, small motors capable of fine control at low speeds were required. A brushed DC gear-motor from Pololu was selected (Figure 3.4). A DC motor is well suited for the application because of its low cost, small size, and simple electrical drive requirements. The selected motor also has provisions for encoder feedback, which is important for good speed control. Pololu sells many versions of this motor, but for this application a low-power version with a 50:1 gear ratio was chosen.

Here in the motor selection trade, it's obvious once more how the small size of the robot is beneficial. These small drive motors use only 40mA running current at 5V, which was helpful later in the design when it came time to specify the power supply and battery systems.

The small size of the robot demands a fully integrated motor controller IC. A Toshiba motor controller was selected to drive both motors. This single IC handles both motors, and has good performance even with motor supply voltages as low as 5V. Its continuous current capacity is well above the anticipated motor stall current, which will ensure its

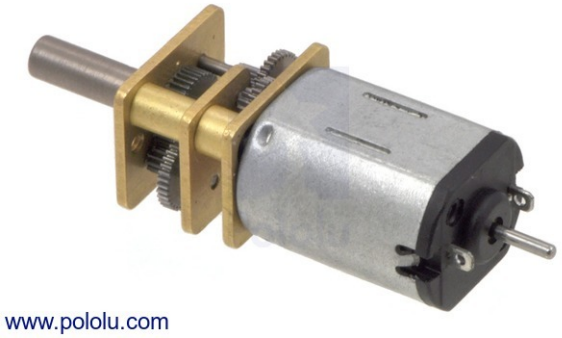


Figure 3.4: Dual-Shaft Gearmotor[1]

survival even under abuse.

An added feature of the motor controller is a hardware disable pin. In this application the pin is routed to a switch on the side of the robot. Using this switch, the operator can bypass the firmware and manually disable the motors. This is helpful during development, to prevent an errant command from sending the robot running away across the floor or table.

In order to achieve wide control bandwidth, a experiment was run to characterize the drive system. Wheel speed was measured in arbitrary units at duty cycles from 0 to 1 on three different surfaces: shag carpet, smooth table, and free (no surface contact). Figure 3.5 shows that the wheel speed on a smooth table is well-approximated over most of the range by a linear curve fit. Even better, Figure 3.6 implies that this relationship is not strongly affected by surface conditions. The slope of this line was used as a feed-forward term in the motor speed control algorithm, and an integrator was used to reduce steady-state error.

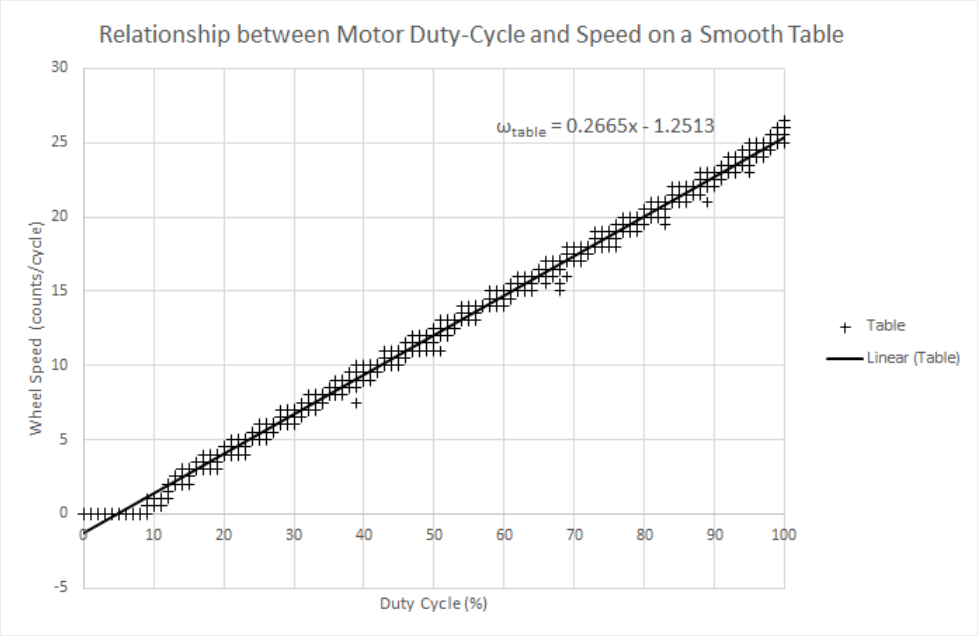


Figure 3.5: Wheel Speed on a Smooth Table

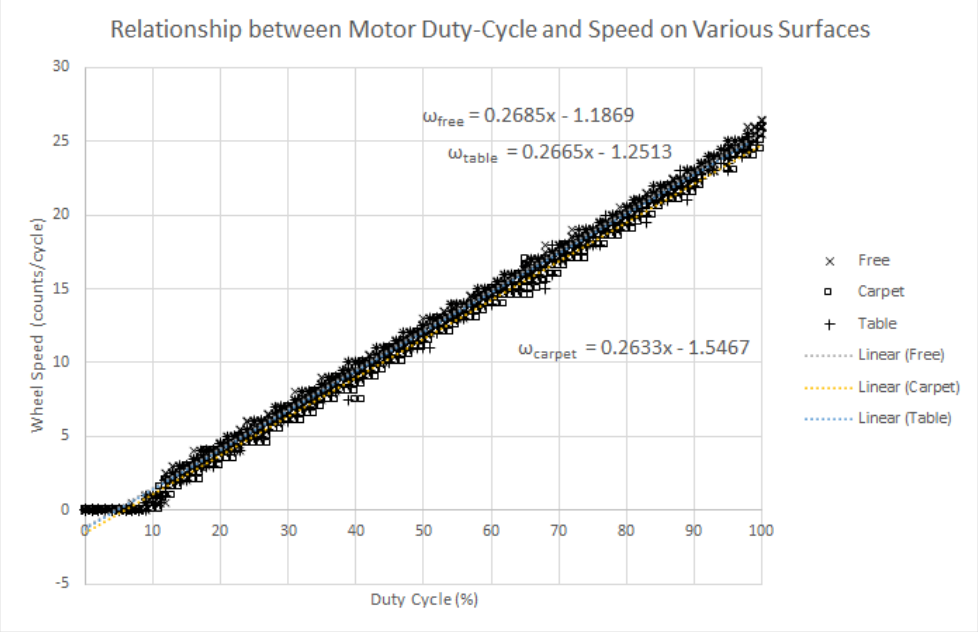


Figure 3.6: Wheel Speed on Various Surfaces

3.6 Communication System

In order to communicate commands from the control computer to robot, a wireless system is required. Several systems were considered, including IR, but eventually an XBee module was selected. The XBee module's primary advantages are its high level of integration and ease of use. A pair of XBee modules can be configured to act as a transparent serial connection, essentially mimicking a direct RS-232 connection between the PC and the robot. Furthermore, a collection of XBee modules can all be configured to receive a broadcast message from a single transmitter. This 1-to-N feature was used to send commands to multiple robots, with each robot selectively filtering out commands which are not intended for it specifically.

In order for each robot to know its identity, a bank of 3 dip switches is attached to the underside of the robot. Using these switches, an operator can configure the address of each robot as a value from 0 to 7. Commands from the control computer contain a "Destination ID" field which is used by the robot firmware to determine whether to act on or discard the message.

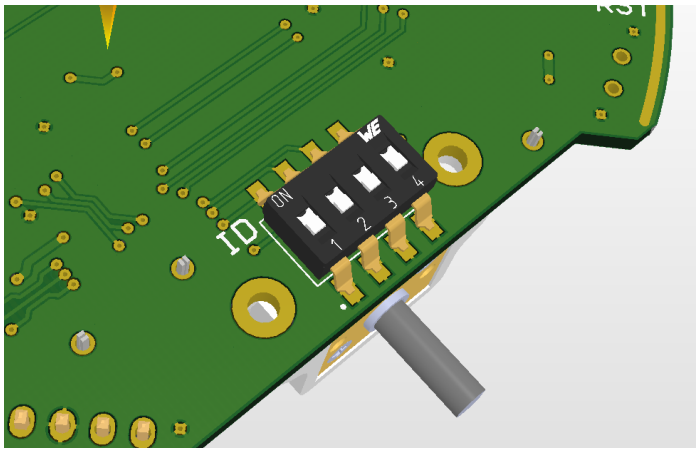


Figure 3.7: DIP Switches to set Robot ID. Fourth position unused.

The firmware in the robot is configured to read incoming messages one byte at a time, at whatever speed they arrive. Start and stop characters help the firmware bound the

contents of an individual message.

A typical message such as the Movement Command consists of 7 ASCII bytes:

#ABBCC*

where the value of each byte/field is described in Table 3.2.

Field		**Value**
#	Start Character	"#"
A	Destintaion ID	"0"–"7"
BB	Right Motor Command	"00"–"99"
CC	Left Motor Command	"00"–"99"
*	Stop Character	"*"

Table 3.2: Movement Command Structure

At the time of writing, the effective range of motor speed commands is approximately -24 to 24, based on the firmware update rate and encoder pulses-per-revolution. In order to enable reverse motor commands without having to transmit a sign character, motor speed commands are offset by 30 counts. That is, a command of 30 is interpreted by the firmware as 0, and commands of less than 30 are interpreted as negative values. Future work on the robot firmware could change how this interface works, but presently that's beyond the scope of this paper.

Other message types are possible as well. As a proof of concept, the robot also supports a command which can be used to modify the behavior of the onboard speaker. Future applications could implement more message types in order to achieve specific robot behaviors.

3.7 Power System

In order to design the power system, two studies need to be done: a power budget and an energy budget. The power budget needs to take into account the various voltages that need to be delivered, and the maximum current expected to be required by those systems. The energy budget is an extension of this, in which estimated duty cycles and voltage conversion efficiencies are included in order to develop an estimate of the required battery capacity.

Figure 3.8 shows the overall architecture of the power system. The following sections describe how this architecture was developed and how the individual parts were chosen.

3.7.1 Power Budget

The design of the robot to this point includes components which require both 3.3V and 5V power. Table 3.3 lists the robot subsystems and how much power they are expected to draw in a worst-case scenario. These values were obtained from the relevant datasheets, in conjunction with the required circuitry for this use case. The last line in Table 3.3 shows what we expect to be the maximum worst case current draw on each of the two voltage rails. This information will feed into the selection of DC-DC voltage converters which condition the power from the battery and supply it to the subsystems at the correct voltages.

Item	$V_{3.3}I_{max}$	$V_5 I_{max}$
Motors		600mA
Buzzer	3mA	
LEDs	8mA	
xBee	50mA	
I2C Bus	7mA	
Microcontroller	30mA	
Totals	98mA	600mA

Table 3.3: Power Budget

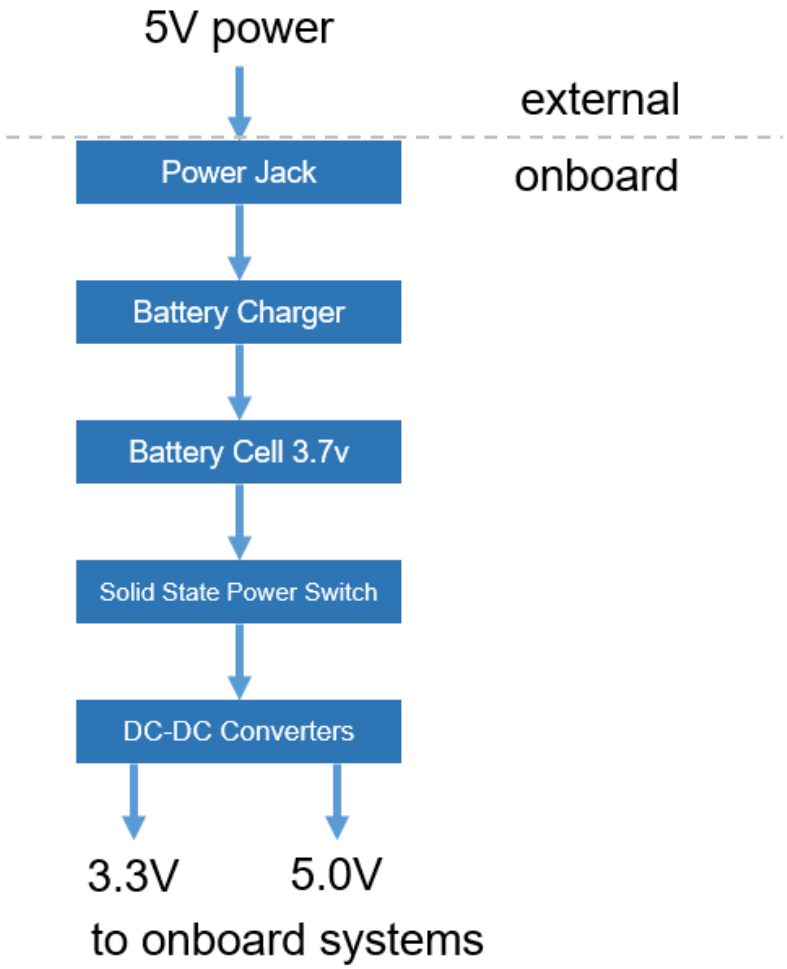


Figure 3.8: Diagram of the robots' onboard power system

3.7.2 Voltage Converters

It's been established that the robot needs two power rails: 3.3V and 5V. The LiPo cell delivers somewhere between 4.2V and 3V depending on conditions, so an active system is required to convert this to a regulated voltage suitable for the robot subsystems.

Two converters were selected to produce the required 5V and 3.3V supplies. Their input specifications cover the entire operating range of the LiPo cell, and their current output capabilities meet the requirements described in Table 3.3. Figure 3.9 describes the efficiency of the units over a range of operating conditions. In this application the converters are expected to be operating at approximately 3.7V input, which puts them somewhere around 85% efficiency[2][4].

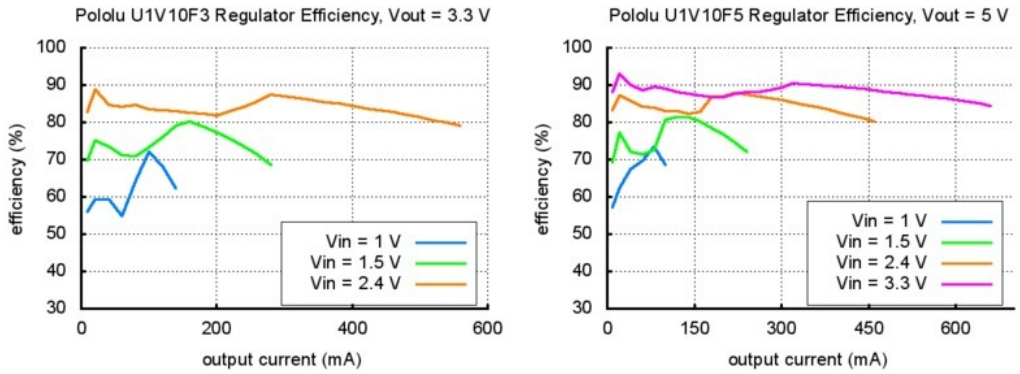


Figure 3.9: Selected Voltage Converter Efficiencies[2][4]

3.7.3 Energy Budget

Table 3.4 lists the robot subsystems and how much power they are expected to draw in on average, with the robot moving and communicating.

Item	$V_{3.3}I_{avg}$	V_5I_{avg}
Motors		100mA
Buzzer	1mA	
LEDs	2mA	
xBee	50mA	
I2C Bus	7mA	
Microcontroller	30mA	
Totals	90mA	100mA

Table 3.4: Energy Budget

The input current on the converter is approximated by

$$I_{in} = \frac{V_{out}I_{out}}{\eta V_{in}} \tag{3.1}$$

where η is the converter efficiency - 85% - as established from Figure 3.9. Assuming a nominal cell voltage of 3.7V,

$$I_{in3.3} = \frac{V_{out}I_{out}}{\eta V_{in}} = \frac{3.3 * 90}{0.85 * 3.7} = 94mA \tag{3.2}$$

$$I_{in5} = \frac{V_{out}I_{out}}{\eta V_{in}} = \frac{5 * 100}{0.85 * 3.7} = 159mA \tag{3.3}$$

Summing the results of equations 3.2 and 3.3 gives

$$I_{batt} = I_{in3.3} + I_{in5} = 94 + 159 = 253mA \tag{3.4}$$

The target battery life of the robot is in the range of a few hours. A 1000mA hour cell was chosen to meet these requirements while also being small enough to fit onboard. The battery was placed below the PCB in order to lower the center of gravity.

3.7.4 Solid State Power Switch

A miniature slide switch was chosen as the power control for the robot. Unfortunately, it's difficult to find such a small switch which is rated for the maximum current expected to drawn from the battery. It was decided to use the slide switch to control a solid-state power switch instead. The slide switch is used to control the gates on a pair of common-source MOSFETs which serve as a solid state relay to interrupt the connection between the battery cell and the voltage converters. This circuit is on the low side of the battery interface, because the MOSFETs require positive gate voltage to turn on. Incidentally, because the MOSFET requires a positive V_{gs} to turn on, this also functions as a polarity protection circuit which would prevent damage in the event of a mis-connected battery.

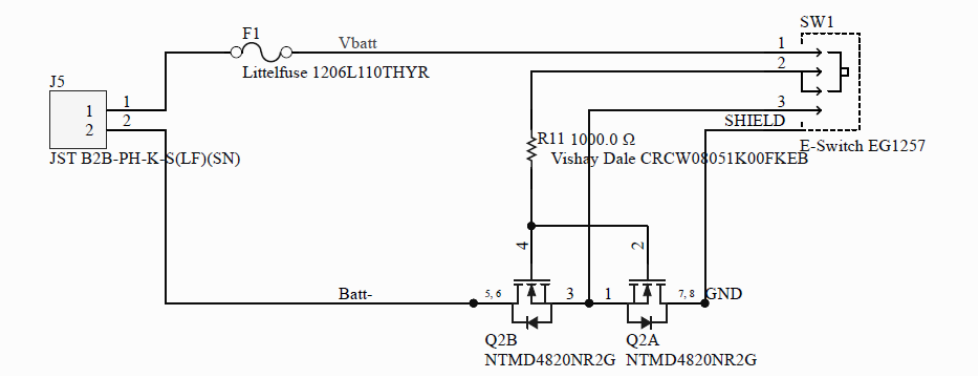


Figure 3.10: Common-Source Power Switch

3.7.5 Battery and Charger

In order to power the robot, a single cell Lithium Polymer battery was selected. LiPo cells are easy to charge using off-the-shelf ICs, and provide excellent power and energy density in terms of both weight and volume.

Charging LiPo battery cells should be done carefully, as overcharging can lead to thermal runaway. In order to mitigate this risk, a purpose-built off-the-shelf IC was chosen to perform the battery charging. This IC accepts 5V power and uses it to charge the cell.

It takes the cell through a multi-stage charge process which ends in a maintinence mode suitable for keeping the battery at full charge safely. By integrating this IC onto the robot itself, the need for external chargers or systems is elimainted. Only a 5V power supply is required to charge the robot. A 5.5mm barrel jack was chosen as the power connector, primarly for its ubiqity and durrability. A micro-USB was also considered, but the idea was discarded because the connector was considered too fragile and too difficult to solder reliably by hand.

Battery charge status indication is accomplished via a pair of LEDs located near the power jack. An amber LED indicates that charging is in progress, and a green LED indicates that the cell has reached full charge.

Because the solid state power switch does not interrupt the connection between the battery charger and the power jack, it's possible to charge the battery while the robot is turned off.

3.8 Capability Growth Features

In anticipation of future experiments, the design of the robot incorporates two opportunities for growth: optical time-of-flight distance measurement modules, and a generic expansion interface.

The optical time-of-flight distance measurement interface is actually a series of three connection positions, located on the front of the robot. Each position can accommodate a high speed, high accuracy, optical time-of-flight range-finder capable of resolving objects of various colors/reflectivities to within 1mm[5]. The modules communicate with microcontroller using I^2C and fit entirely within the existing physical envelope of the robot.

The generic expansion interface is a 14-pin connector capable of various digital, analog, and communication functions with the microcontroller. It's designed to allow the addition of more complex features such as local sensor suites, on-board displays, or alternative

interface controllers.

Experimental Results

4.1 Calibration and Verification

With a physical system it's impossible to achieve zero error, but we can at least measure and quantify the error, and potentially determine whether that error is large enough to be significant in the context of a given experiment.

Key parameters that the testbed intends to measure or estimate include:

- Robot Position
- Robot Speed/Velocity
- Robot Orientation or Angular Position
- Robot Angular Velocity

4.1.1 Pixel Grid Scale Calibration for Position and Speed

By imaging items of known physical dimensions and calculating their dimensions in pixels, we can develop a scale factor which relates pixels to millimeters. A calibration device, shown in Figure 4.1, of known dimensions was built and imaged in various positions within the plane. Performing this measurement several times at various locations around the test bed, we can calculate the experimentally-determined pixel scale constant:

$$k_p = 0.565 \qquad \text{pix/mm}$$

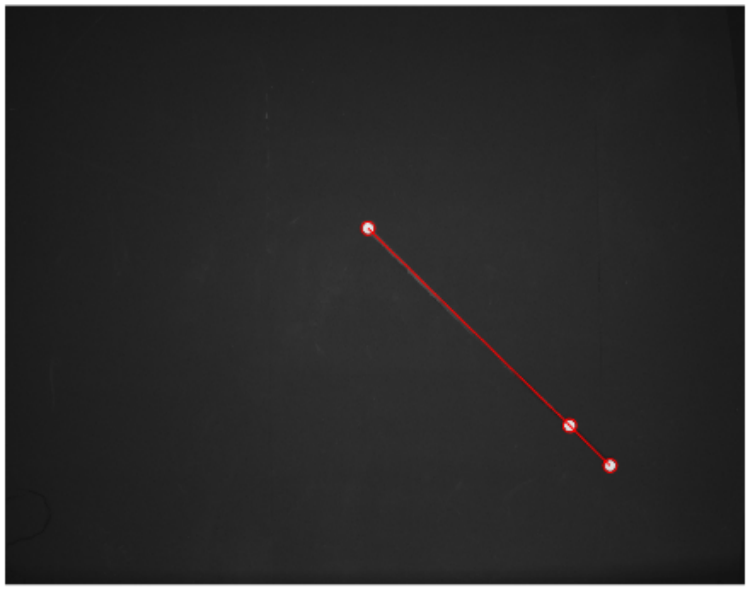


Figure 4.1: Linear Calibration Tool, with dots at known distances from each other

Once we have this value, we can begin to evaluate our robot speed control and measurement. An experiment was performed where the robot was commanded to drive open-loop in a rough circle, at a slowly increasing speed. This path is shown in Figure 4.2. The actual path isn't significant; it's shape is just to keep the robot within the field of view for long enough to collect a continuous dataset.

Robot speed S_R can be calculated using position and time deltas between camera frames:

$$S_R[t] = \frac{\sqrt{(x[t] - x[t - dt])^2 + (y[t] - y[t - dt])^2}}{dt} \tag{4.1}$$

Where dt is measured loop time, nominally 50ms. Speed commands are sent to the robot as a unitless number of "counts" which relates to how many encoder pulses per second the robot is controlling to. A conversion is required to put the value into more un-

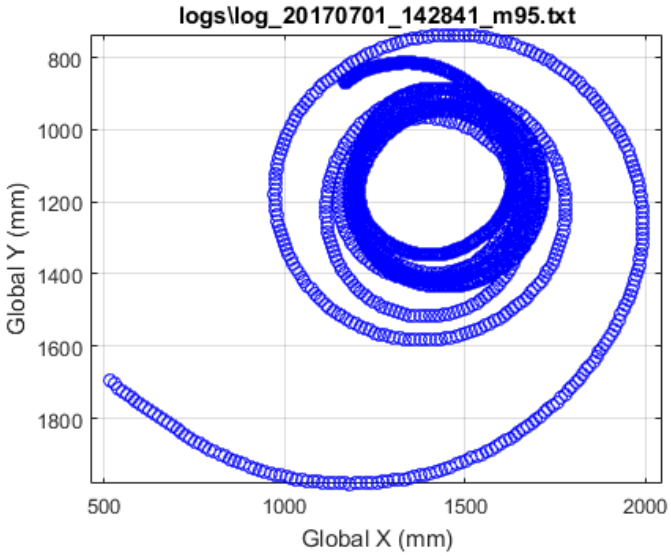


Figure 4.2: Open-loop robot speed test path

derstandable units. The theoretical relationship between speed command and actual speed is

$$K_{sT} = \frac{D_W \pi K_F}{K_E K_G} \tag{4.2}$$

where D_W is the diameter of the wheel, K_E is the number of pulses per encoder rotation, K_G is the gear ratio, and K_F is a constant within the firmware. Setting these values gives us a command slope of

$$K_{sT} = 13.13 \qquad \text{(mm/s/command)} \tag{4.3}$$

with an uncertainty of at least 2% due to wheel diameter, which is a measured value. If we apply equation 4.1 to our measured data set and plot the results in Figure 4.3, we can run a linear regression to get an experimental result:

$$K_{sE} = 13.47 \qquad \text{(mm/s/command)} \tag{4.4}$$

The results demonstrate that there is less than 3% error between K_{sT} and K_{sE}.

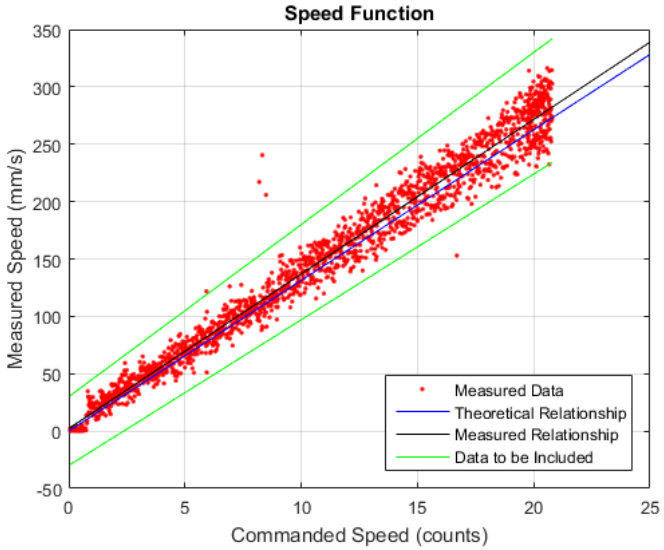


Figure 4.3: Robot speed as measured, compared to theoretical speed across command range

4.1.2 Angular Position and Velocity Calibration

An experiment was performed where the robot was commanded to spin in place, at a slowly increasing rate. The measured orientation of the robot was recorded at each time-step and plotted vs the command in Figure 4.4. The stair-step appearance of the data is due to limits in the granularity to which the firmware can control wheel speed. This is not an insurmountable problem; it's correctable by switching the wheel speed measurement within the firmware from pulses-per-cycle to ticks-per-pulse. The experiments presented here were designed to work within the constraints of this granularity, but making that improvement to the firmware and repeating this analysis would be a valuable and obvious follow-on effort.

The theoretical $K_{\omega T}$ which relates command value to actual angular velocity can be calculated as

$$K_{\omega T} = \frac{K_{sT}}{W_B} \qquad \text{(rad/s/command delta)} \tag{4.5}$$

where W_B is wheel base, in this case 78mm, and K_{sT} is the speed constant we previously calculated in equation 4.2. The command delta is simply the difference in command between the two wheels. In particular, W_B has an uncertainty of perhaps 1-2% due to

assembly variations and measurement error. Filling in all the variables we arrive at:

$$K_{\omega T} = 0.1683 \qquad \text{(rad/s/command delta)} \tag{4.6}$$

The experimental value is arrived at by performing a linear fit on the data, shown in Figure 4.4.

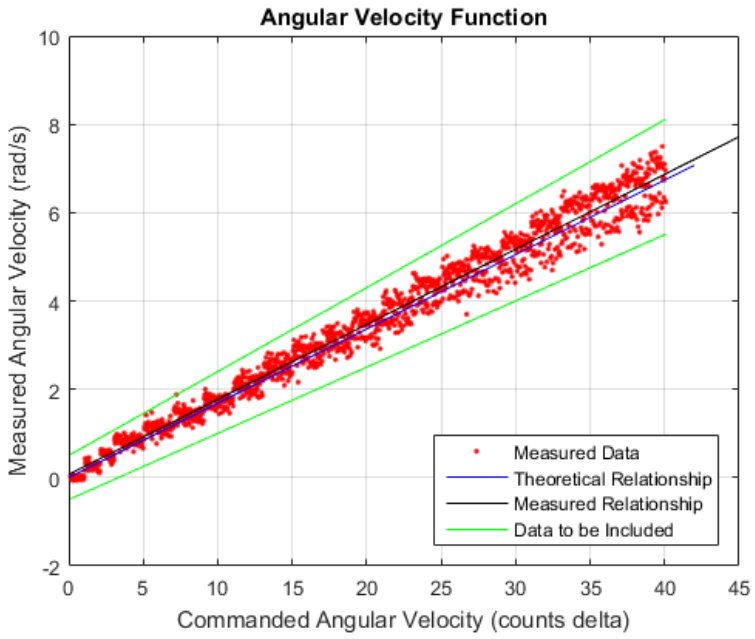


Figure 4.4: Robot angular velocity as measured, compared to theoretical across command range

$$K_{\omega E} = 0.1698 \qquad \text{(rad/s/command delta)} \tag{4.7}$$

Comparing $K_{\omega E}$ and $K_{\omega T}$ we see that they're within about 1% which, given the uncertainties, is as good as we can expect.

4.2 Dubins Path

Having verified the robots' performance and generated a set of constants with which to relate commands to real-world-behaviors, it's now possible to run a control algorithm scaled in real-world-units. A previously-developed Dubins Cone path solver was inserted into the MATLAB code, with its control parameters set to match those of the physical robot. This solver requires the following inputs:

- Vehicle and Target State (Position and Orientation of each)
- Vehicle Speed (a constant)
- Vehicle maximum angular velocity

The solver finds the shortest route from the Vehicle position to the Target position, with the added constraint that the Vehicle orientation must match the target orientation at the time of arrival. The Target is assumed to be stationary. In simpler terms, this problem is not unlike an airplane plotting a course to a landing strip; it can't stop moving forward, it can only turn so fast, and it needs to be facing the right way when it gets there.

The solver produces a sequence of commands which describe which way the Vehicle should turn, for how long, etc in order to reach the Target. In an ideal setup the initial path could be followed directly, open loop, and the actual path would match it perfectly. Instead of relying on open loop control, we'll instead recalculate the optimal path at every timestep, 50ms, and adjust course as required. The solver produces the entire path solution every time you run it, but in our case we'll be using only the initial command for each solution. Our software will plot the entire projected path at every timestep so we can see what the solver is doing in real time. Since we're running the solver repeatedly in closed loop, we'll call it a "controller" from here forward.

A problem arises from the fact that this particular controller is always calling for a turn, a section of straight motion, and then another turn. Even if the vehicle's angular

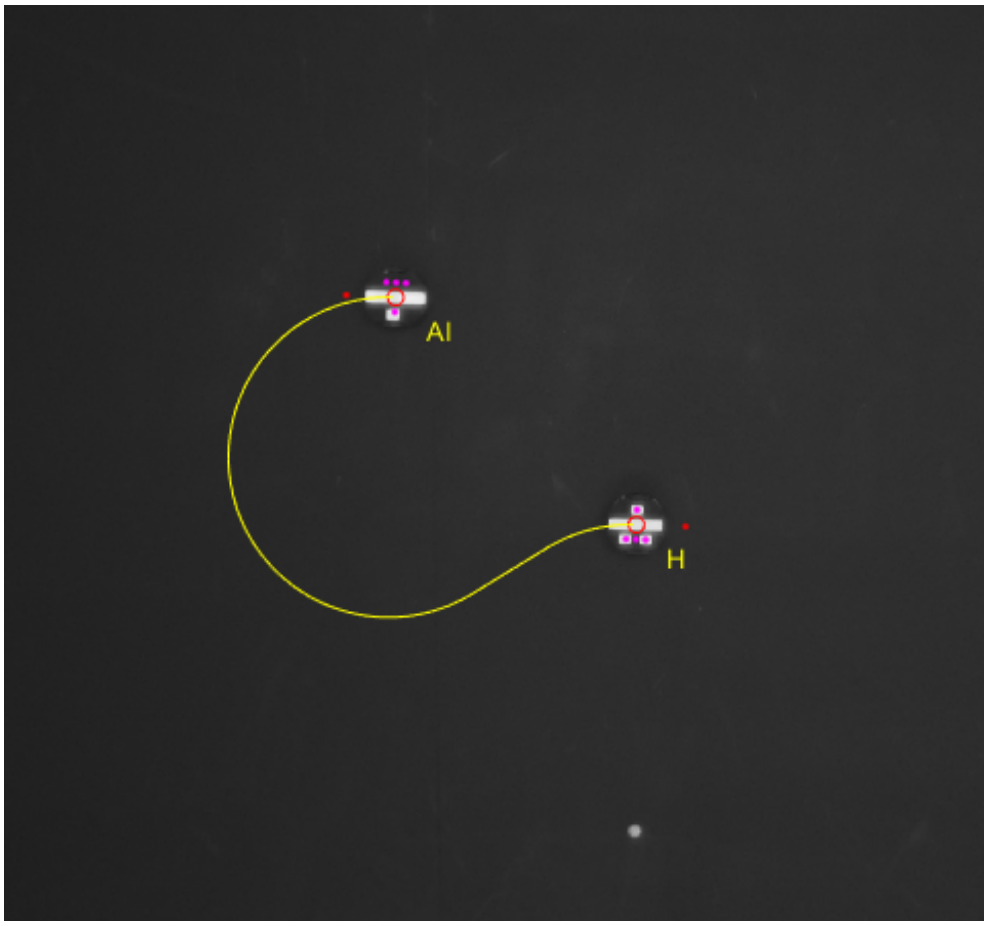


Figure 4.5: Dubins Path, as calculated and projected in real time

position is within 1 arcsecond of correct, the controller will still call for a turn of perhaps 1ms in duration in order to correct it. Since the testbed operates in 50ms discrete time chunks, this 1ms command turns into a 50ms command and the vehicle overshoots the target angle. This causes a small oscillation around the target angle at several Hz.

In order to address this issue the testbed takes commands which have short time durations and reduces their amplitude. Empirical testing demonstrates that the best behavior is achieved when commands are normalized to a 200ms timeframe. That is, a turn command of 1 (100%) with an estimated duration of 100ms becomes a command of 0.5 (50%). Presumably this 200ms command normalization helps alleviate the symptoms of a few different types of delay (image processing, wheel acceleration, etc), but a more thorough analysts isn't available at this time. In the context of this paper, the empirical results are

sufficient in so much as any side-effects of command normalization are less apparent than the issues that arise without it.

4.2.1 Delta between Expected and Implemented Angular Velocities

One interesting behavior which became apparent is demonstrated in Figure 4.6. Let ρ_E be the controller's expectation of its capability for angular velocity, and ρ_I be the vehicle's actual implemented capability for angular velocity. In this case, the vehicle wasn't capable of the ρ that the controller was expecting. That is, $\rho_I < \rho_E$.

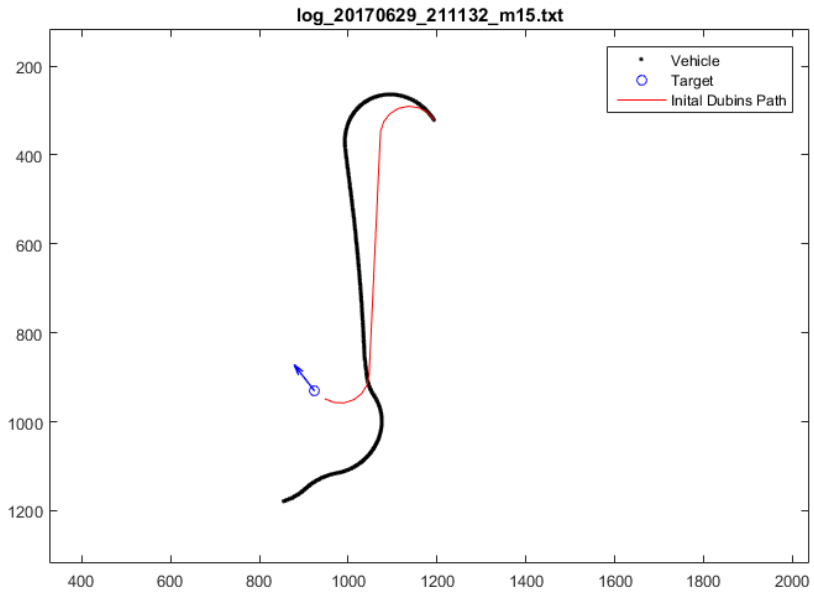


Figure 4.6: Actual and Initially-Calculated Paths with $\rho_I < \rho_E$

In the $\rho_I < \rho_E$ case, the robot takes longer than anticipated to complete the first turn. It calculates a new, direct route to the location of the next anticipated maneuver and arrives there correctly. At this point, something goes wrong. The robot inadvertently moves beyond the turn radius line and the controller is forced to calculate a new route. At first glance it might be assumed that this is a consequence of the $\rho_I < \rho_E$ condition, in so much

as it's obvious that the robot wasn't going to be able to make that final turn. As it turns out however, the $\rho_I > \rho_E$ situation produces a similar but less pronounced behavior. See Figure 4.7.

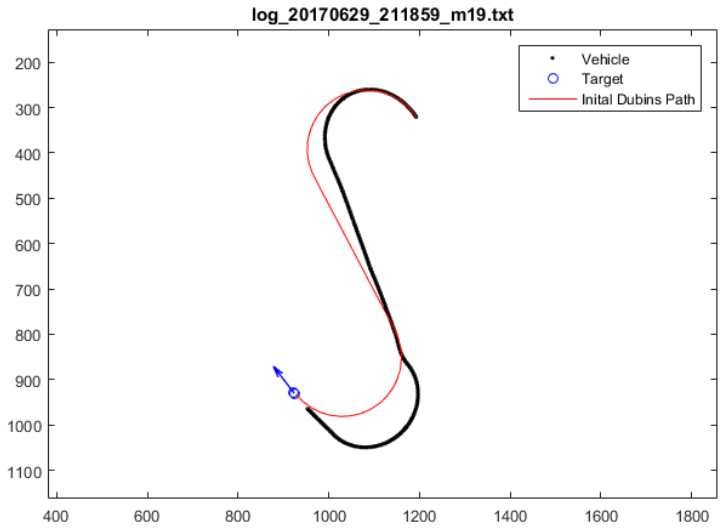


Figure 4.7: Actual and Initially-Calculated Paths with $\rho_I > \rho_E$

In this case the robot completes the first turn sooner than expected, and calculates a new shorter path to what it expects is the location to begin the second turn. This is not a problem for the controller, in that it completes the maneuver in shorter than the originally anticipated time and seems to be on track to complete the path. As it begins the second turn however, it unintentionally crosses over the line of minimum turn radius by a small amount. This causes the current trajectory to be suddenly unsuitable from the perspective of the controller, and an entirely new path with a wider turn is calculated instead.

So was this behavior caused by the $\rho_I \neq \rho_E$ situation, or something else? Further testing shows that under these initial conditions, something like this happens nearly every time regardless of the relationship between ρ_I and ρ_E. In fact, it's even possible to reproduce this behavior in simulation by adding delay to the simulated feedback. Because the con-

troller is solving for the minimum path, the solution is necessarily on the edge of stability. Any small deviation across the line of stability causes the controller to abort and calculate an entirely new path. Since the controller is running in discrete time, and the real system in running in continuous time, there will always be small deviations.

What we find with these experiments is under certain conditions this particular controller is very sensitive to small deviations from the projected path. This is a natural consequence of the controller choosing the absolute minimum time/length path, with no provisions for the kind of errors that can occur in a system like this one.

Trying to "*fix*" this controller is tempting, but is beyond the scope of this paper and would represent an entirely new effort with different goals. The goal here is only to demonstrate the performance of the testbed and robots, make some observations, and give examples of possible use-cases for the system.

4.3 More Complex Implementations

Two robots both using the Dubins Controller were set to chase each other (Figure 4.8). It was observed that with equal performance parameters, they will never catch each other. In fact, they quickly fall into a stable relative configuration and stay that way.

Two robots were set to chase a third, manually controlled robot (Figure 4.9). The testbed is capable of tracking and controlling up to 8 robots at a time, and applying different control algorithms to any of them. At the time of writing only 4 robots have been constructed.

In this case it became apparent that it was possible to "trick" the controller(s) into disadvantageous behavior. For example, manipulating the state so that two of the robots collide, or one of them leaves the playing area. This may be a limitation of a simple controller, but it's also behavior which may have been tedious to predict, create, and observe in a simulated environment. Using the testbed it was possible to casually discover, observe,

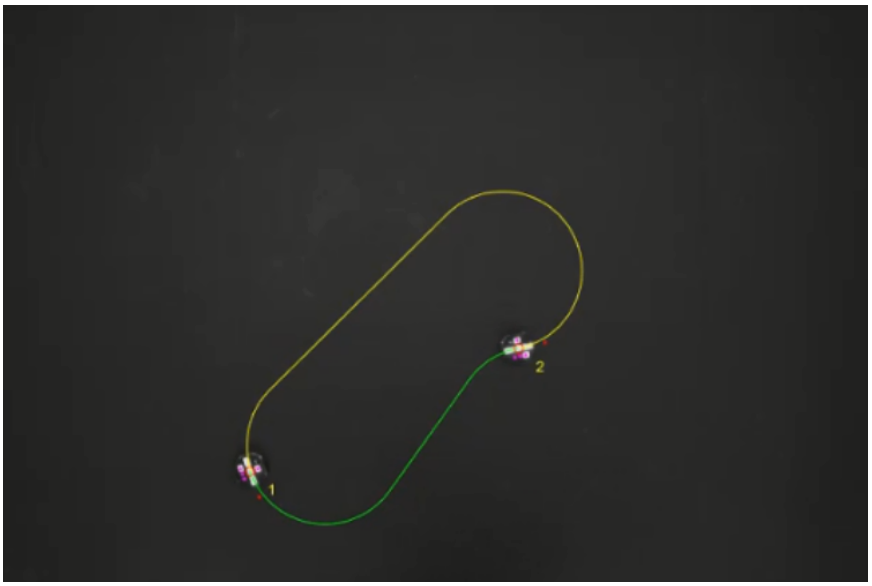


Figure 4.8: Two robots chasing each other using the Dubins Controller

and record the behavior all in a matter of minutes. Evaluations of more complex controllers would benefit from this real-time interaction in a similar way.

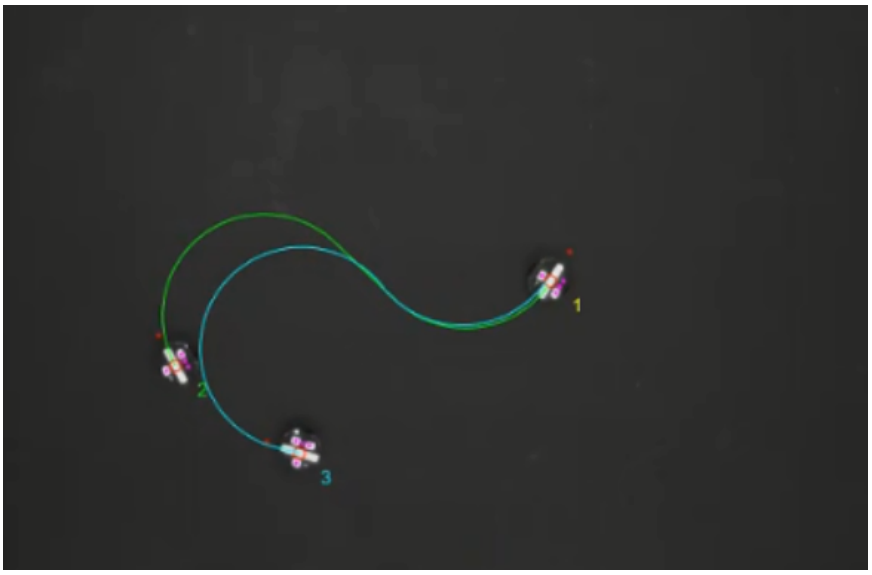


Figure 4.9: Two robots both chasing a third, manually operated robot

In this example, one robot chases another which in turn chases a third. By manually controlling one of the robots, it's easy to manipulate the state of the overall system and observe the subsequent behaviors. In this case the manually controlled robot is not con-

strained by the typical rules of a Dubins vehicle. It can stop, turn in place, and even reverse. It's also three times as fast as the autonomous vehicles, which gives the human more control over the situation. All of these parameters are easily controlled from the MATLAB code, in order to configure the testbed for whatever experiment is required.

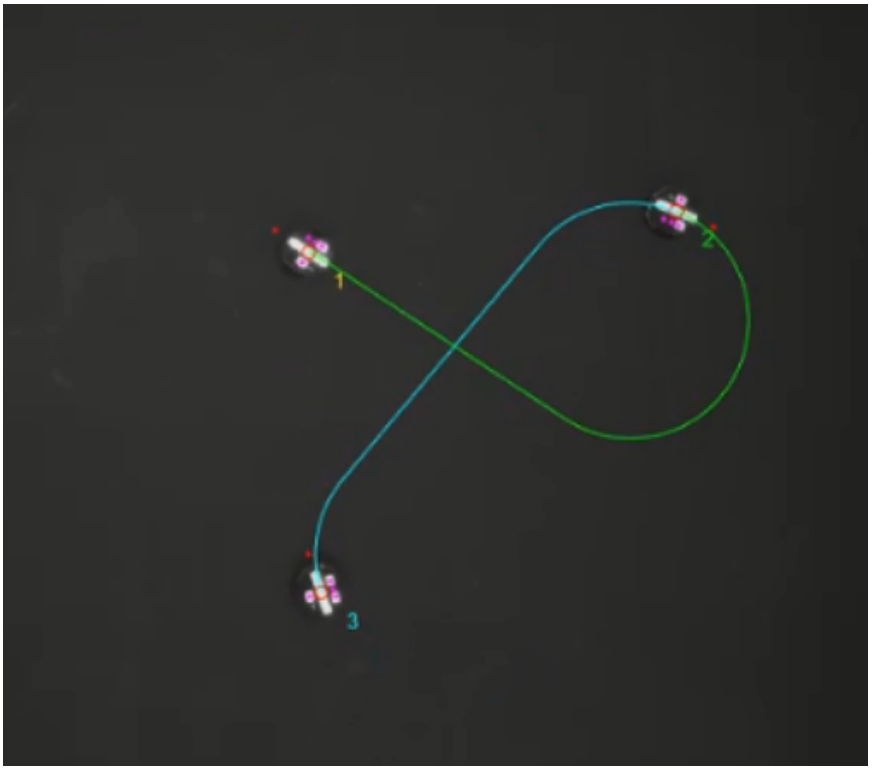


Figure 4.10: Robot chasing another which in turn chases a third

4.4 Conclusions

An physical multi-agent autonomy testbed was designed and constructed. After a simple calibration, testing shows that the system accurately reproduces the overall behavior of a simulated controller. As hypothesized, small errors in the real system can cause noticeable changes in overall controller behavior under certain conditions, and seeing this happen helps highlight potential regions of control instability. By interacting with the controller in real time, it becomes apparent that it might be possible to trick the controller into disadvan-

tageous behavior in some cases. Future experiments with more complex controllers should provide proportionally greater insight into their strengths and weaknesses.

Bibliography

[1] Pololu Corporation. *50:1 Micro Metal Gearmotor LP 6V with Extended Motor Shaft*. June 24, 2017. URL: https://www.pololu.com/product/2203.

[2] Pololu Corporation. *Pololu 3.3V Step-Up Voltage Regulator U1V10F3*. June 2017. URL: https://www.pololu.com/product/2563.

[3] Pololu Corporation. *Pololu 3pi Robot*. June 2017. URL: https://www.pololu.com/product/975.

[4] Pololu Corporation. *Pololu 5V Step-Up Voltage Regulator U1V10F5*. June 2017. URL: https://www.pololu.com/product/2564.

[5] Pololu Corporation. *VL6180X Time-of-Flight Distance Sensor Carrier*. July 2017. URL: https://www.pololu.com/product/2489.

[6] MathWorks. *cameraParameters class*. July 2017. URL: https://www.mathworks.com/help/vision/ref/cameraparameters-class.html.

[7] Eric Nees. *Robot G8*. July 2017. URL: https://circuitmaker.com/Projects/Details/eric-nees-3/Robot-G8.

[8] Timothy J. Pennings. "Do Dogs Know Calculus?" In: *College Mathematics Journal*. Vol. 34. May 2003, pp. 178–182.

[9] RoadNarrows Robotics. *RoadNarrows Robotics*. July 2017. URL: https://roadnarrows.com/.

[10] Seeed Studio. *Seeed Studio*. July 2017. URL: https://www.seeedstudio.com/.

Appendix A: Microcontroller

PIC32MX1XX/2XX 28/36/44-PIN

32-bit Microcontrollers (up to 256 KB Flash and 64 KB SRAM) with Audio and Graphics Interfaces, USB, and Advanced Analog

Operating Conditions

- 2.3V to 3.6V, -40ºC to +105ºC, DC to 40 MHz
- 2.3V to 3.6V, -40ºC to +85ºC, DC to 50 MHz

Core: 50 MHz/83 DMIPS MIPS32® M4K®

- MIPS16e® mode for up to 40% smaller code size
- Code-efficient (C and Assembly) architecture
- Single-cycle (MAC) 32x16 and two-cycle 32x32 multiply

Clock Management

- 0.9% internal oscillator
- Programmable PLLs and oscillator clock sources
- Fail-Safe Clock Monitor (FSCM)
- Independent Watchdog Timer
- Fast wake-up and start-up

Power Management

- Low-power management modes (Sleep and Idle)
- Integrated Power-on Reset and Brown-out Reset
- 0.5 mA/MHz dynamic current (typical)
- 44 µA I_{PD} current (typical)

Audio Interface Features

- Data communication: I^2S, LJ, RJ, and DSP modes
- Control interface: SPI and I^2C™
- Master clock:
  - Generation of fractional clock frequencies
  - Can be synchronized with USB clock
  - Can be tuned in run-time

Advanced Analog Features

- ADC Module:
  - 10-bit 1.1 Msps rate with one S&H
  - Up to 10 analog inputs on 28-pin devices and 13 analog inputs on 44-pin devices
- Flexible and independent ADC trigger sources
- Charge Time Measurement Unit (CTMU):
  - Supports mTouch™ capacitive touch sensing
  - Provides high-resolution time measurement (1 ns)
  - On-chip temperature measurement capability
- Comparators:
  - Up to three Analog Comparator modules
  - Programmable references with 32 voltage points

Timers/Output Compare/Input Capture

- Five General Purpose Timers:
  - Five 16-bit and up to two 32-bit Timers/Counters
- Five Output Compare (OC) modules
- Five Input Capture (IC) modules
- Peripheral Pin Select (PPS) to allow function remap
- Real-Time Clock and Calendar (RTCC) module

Communication Interfaces

- USB 2.0-compliant Full-speed OTG controller
- Two UART modules (12.5 Mbps):
  - Supports LIN 2.0 protocols and IrDA® support
- Two 4-wire SPI modules (25 Mbps)
- Two I^2C modules (up to 1 Mbaud) with SMBus support
- PPS to allow function remap
- Parallel Master Port (PMP)

Direct Memory Access (DMA)

- Four channels of hardware DMA with automatic data size detection
- Two additional channels dedicated for USB
- Programmable Cyclic Redundancy Check (CRC)

Input/Output

- 10 mA source/sink on all I/O pins and up to 14 mA on non-standard V_{OH}
- 5V-tolerant pins
- Selectable open drain, pull-ups, and pull-downs
- External interrupts on all I/O pins

Qualification and Class B Support

- AEC-Q100 REVG (Grade 2 -40ºC to +105ºC) planned
- Class B Safety Library, IEC 60730

Debugger Development Support

- In-circuit and in-application programming
- 4-wire MIPS® Enhanced JTAG interface
- Unlimited program and six complex data breakpoints
- IEEE 1149.2-compatible (JTAG) boundary scan

Packages

Type	SOIC	SSOP	SPDIP	QFN		VTLA		TQFP
Pin Count	28	28	28	28	44	36	44	44
I/O Pins (up to)	21	21	21	21	34	25	34	34
Contact/Lead Pitch	1.27	0.65	0.100"	0.65	0.65	0.50	0.50	0.80
Dimensions	17.90x7.50x2.65	10.2x5.3x2	1.365"x.285"x.135"	6x6x0.9	8x8x0.9	5x5x0.9	6x6x0.9	10x10x1

Note: All dimensions are in millimeters (mm) unless specified.

Appendix B: LiPo Charger IC

MCP73831/2

Miniature Single-Cell, Fully Integrated Li-Ion, Li-Polymer Charge Management Controllers

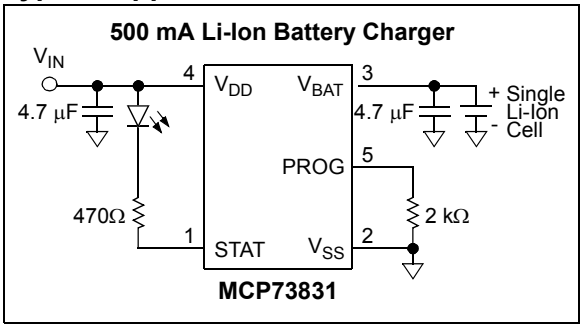
Features:

- Linear Charge Management Controller:
  - Integrated Pass Transistor
  - Integrated Current Sense
  - Reverse Discharge Protection
- High Accuracy Preset Voltage Regulation: ± 0.75%
- Four Voltage Regulation Options:
  - 4.20V, 4.35V, 4.40V, 4.50V
- Programmable Charge Current: 15 mA to 500 mA
- Selectable Preconditioning:
  - 10%, 20%, 40%, or Disable
- Selectable End-of-Charge Control:
  - 5%, 7.5%, 10%, or 20%
- Charge Status Output
  - Tri-State Output - MCP73831
  - Open-Drain Output - MCP73832
- Automatic Power-Down
- Thermal Regulation
- Temperature Range: -40°C to +85°C
- Packaging:
  - 8-Lead, 2 mm x 3 mm DFN
  - 5-Lead, SOT-23

Applications:

- Lithium-Ion/Lithium-Polymer Battery Chargers
- Personal Data Assistants
- Cellular Telephones
- Digital Cameras
- MP3 Players
- Bluetooth Headsets
- USB Chargers

Typical Application



Description:

The MCP73831/2 devices are highly advanced linear charge management controllers for use in space-limited, cost-sensitive applications. The MCP73831/2 are available in an 8-Lead, 2 mm x 3 mm DFN package or a 5-Lead, SOT-23 package. Along with their small physical size, the low number of external components required make the MCP73831/2 ideally suited for portable applications. For applications charging from a USB port, the MCP73831/2 adhere to all the specifications governing the USB power bus.

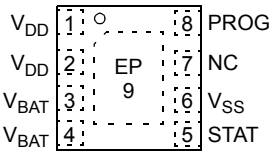
The MCP73831/2 employ a constant-current/constant-voltage charge algorithm with selectable preconditioning and charge termination. The constant voltage regulation is fixed with four available options: 4.20V, 4.35V, 4.40V or 4.50V, to accommodate new, emerging battery charging requirements. The constant current value is set with one external resistor. The MCP73831/2 devices limit the charge current based on die temperature during high power or high ambient conditions. This thermal regulation optimizes the charge cycle time while maintaining device reliability.

Several options are available for the preconditioning threshold, preconditioning current value, charge termination value and automatic recharge threshold. The preconditioning value and charge termination value are set as a ratio or percentage of the programmed constant current value. Preconditioning can be disabled. Refer to **Section 1.0 "Electrical Characteristics"** for available options and the **Product Identification System** for standard options.

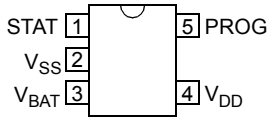
The MCP73831/2 devices are fully specified over the ambient temperature range of -40°C to +85°C.

Package Types

MCP73831/2
2×3 DFN*



MCP73831/2
SOT-23-5



* Includes Exposed Thermal Pad (EP); see Table 3-1.

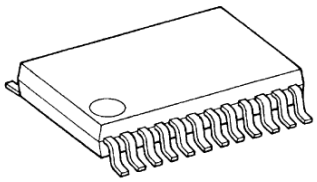
Appendix C: Dual Motor Driver

Toshiba Bi-CD Integrated Circuit Silicon Monolithic

TB6612FNG

Driver IC for Dual DC motor

TB6612FNG is a driver IC for DC motor with output transistor in LD MOS structure with low ON-resistor. Two input signals, IN1 and IN2, can choose one of four modes such as CW, CCW, short brake, and stop mode.



SSOP24-P-300-0.65A

Weight: 0.14 g (typ.)

Features

- Power supply voltage: VM = 15 V(Max)
- Output current: I_{OUT} = 1.2 A(ave)/3.2 A (peak)
- Output low ON resistor: 0.5Ω (upper+lower Typ. @ VM ≥ 5 V)
- Standby (Power save) system
- CW/CCW/short brake/stop function modes
- Built-in thermal shutdown circuit and low voltage detecting circuit
- Small faced package(SSOP24: 0.65 mm Lead pitch)

* This product has a MOS structure and is sensitive to electrostatic discharge. When handling this product, ensure that the environment is protected against electrostatic discharge by using an earth strap, a conductive mat and an ionizer. Ensure also that the ambient temperature and relative humidity are maintained at reasonable levels.